Edgar: Offloading Function Execution to the Ultimate Edge

Technical Report, November 2021, Technische Universität Braunschweig

David Goltzsche
TU Braunschweig, Germany
goltzsche@ibr.cs.tu-bs.de

Tim Siebels
TU Braunschweig, Germany
siebels@ibr.cs.tu-bs.de

Lennard Golsch
TU Braunschweig, Germany
golsch@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig, Germany
rrkapitz@ibr.cs.tu-bs.de

ABSTRACT

Web applications are on the rise and rapidly evolve into mature replacements for their native counterparts. This trend is mainly driven by the attainment of platform-independence and instant deployability. While web applications are getting more and more complex, scalability and responsiveness remain key challenges that are addressed by rather costly approaches such as cloud computing.

In this paper, we present Edgar, a novel middleware for web applications that enables client-side execution of code usually requiring server-side deployment due to missing trust in clients. Following the paradigm of Function-as-a-Service, applications consist of functions that can be distributed to browsers. Other nearby browsers can discover these functions and then directly invoke them on a peer-to-peer basis. Thus, client-side resources are used to provision the web application, which generates lower costs for service providers. Offering premium services such as liberation from ads can be used to incentivise users to provide their resources. In case of resource shortage or unresponsive clients, execution falls back to a cloud-based infrastructure. Edgar combines WebAssembly for executing workloads written in different languages at near-native speed, WebRTC for browser-to-browser communication and Intel SGX to establish trust in other browser's computations. We evaluate Edgar by implementing a digital assistant as well as a recommendation system. Our evaluation shows that Edgar generates lower costs than traditional deployments, scales linearly with increasing client numbers and manages unresponsive clients well.

1 INTRODUCTION

Web applications are becoming the de facto standard for deploying software across various platforms. Advanced web applications even create user experiences similar to native ones by offering features such as offline usage or push notifications while simultaneously liberating users from installing apps. There are multiple reasons for this trend: (i) improved platform-independence through standardised browser APIs; (ii) low entrance barriers for web development; (iii) the possibility of instant deployment of web apps; and (iv) high acceptance by end-users. Web applications need to be scalable to cope with peak demands and highly responsive to offer good user experience.

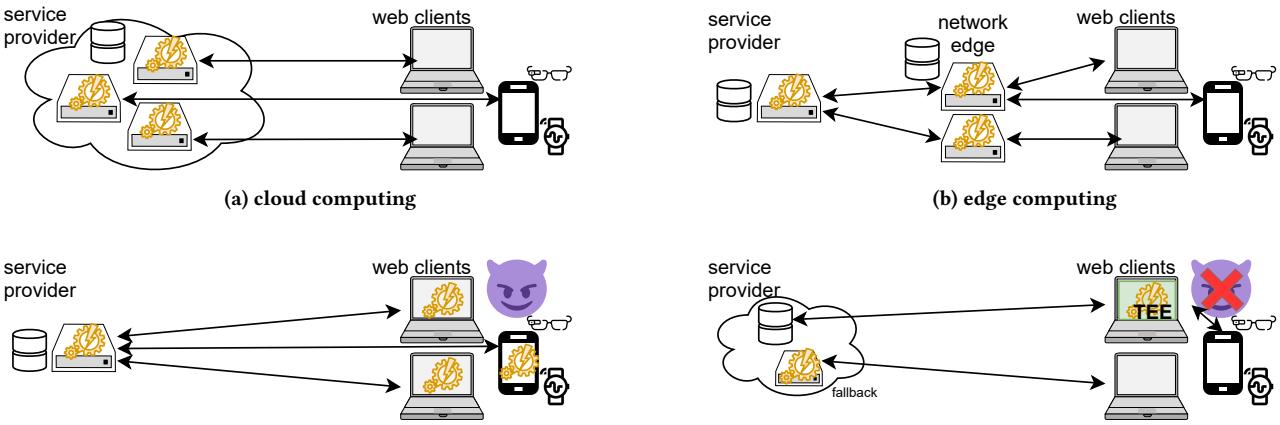
To meet these demands, diverse approaches have been established. Server-side scalability can be improved by deploying applications on highly scalable cloud-based architectures (Fig. 1a). Furthermore, Edge computing [1] and Content Delivery Networks (CDNs) [2, 3] can be used to move computation and data closer to end-users to improve latencies and, therefore, overall responsiveness (Fig. 1b). Although promised to be cost-effective, these cloud-based approaches still come at certain costs [4, 5] which are often underestimated[1]. These costs can even become unsustainable for certain classes of application providers such as non-profit organisations, scientists or start-ups. But also larger companies aim to operate as cost-efficiently as possible.

A more cost-saving approach for service providers is to design their applications to use client-side resources by offloading computation (Fig. 1c). This approach has three advantages: (i) reduced costs for service providers as client-side resources are utilised; (ii) higher responsiveness for users, as less interaction with remote service providers is necessary; and (iii) improved scalability, as the whole application can scale linearly with increasing client numbers. Consequently, outsourcing application logic to web clients is already common practice today. In fact, 98% of websites implicitly use client-side resources by deploying JavaScript code [6]. This is implemented with scripting languages such as JavaScript and –more recently– WebAssembly (see § 2.2), a binary instruction format designed for the web.

However, this offloading approach is generally limited. Web clients are not trustworthy; ultimately, the owners of the machines have to be regarded as powerful attackers. Their attacks can lead to wrong, incomplete or even malicious results or leak confidential code or data. Therefore, application developers so far refrain from offloading sensitive application parts to clients (e.g. code that contains business secrets and/or processes private data). Furthermore, web clients can only perform computations for themselves which inhibits the deployment of distributed applications using client-side resources. Making client-side resources available to other clients can be especially relevant for battery-powered clients with less

[1] https://www.zdnet.com/article/cloud-computing-more-costly-complicated-and-frustrating-than-expected-but-still-essential/

https://doi.org/10.24355/dbbs.084-202111031112-0



(a) cloud computing

(b) edge computing

(c) limited traditional offloading

(d) offloading with EDGAR

Figure 1: Approaches for improving scalability and responsiveness of web applications by moving computations (🧩) to different locations (a-c). EDGAR (d) offloads computation to web clients and uses trusted execution environments (TEEs) to protect against attackers. Computations can still be performed on a server-side fallback for bootstrapping applications.

capable CPUs. Such devices (e.g. wearables) cannot be used for extensive computations, but could benefit from more capable clients with potentially idle resources. Users of these machines can be incentivised to provide resources by offering premium services to them (e.g. ad-free services or bonus features).

To further develop client-side computing in web applications, we present **EDGAR** (Fig. 1d), a novel middleware that enables service providers to distribute their web application equitably over its currently active users. EDGAR opens up new possibilities of client-side resource usage, and can therefore reduce costs for service providers while still offering good scalability and responsiveness. EDGAR achieves this via the following contributions:

(1) Trusted Execution in Web Browsers. EDGAR is the first system that enables trusted execution of both JavaScript and WebAssembly in web browsers. This allows deploying code on client-side that would normally be located on server-side due to confidentiality of the code itself or the data processed by it. This creates new use cases for offloading code and can thus save costs for service providers.

(2) Extended Usage of Client-side Resources. EDGAR extends the concept of using client-side resources in web applications. In EDGAR, clients implicitly provide resources for provisioning the currently used web application *to other users*. By combining this with its trusted execution capabilities (1), EDGAR does not expose code or data and ensures genuine computations. Moreover, it is designed to not overload peers and also enables users to limit resource usage on their machines.

(3) Proximity-based Code Distribution. For achieving contribution (2), a programming model supporting flexible code distribution is needed. Therefore, EDGAR transfers the paradigm of Function-as-a-Service (FaaS) to web clients. This allows the distribution of small, self-contained *functions* based on browser proximity. EDGAR enables the discovery of participating browsers nearby and allows other clients to invoke functions directly without additional network hops. This reduces response times and can therefore improve the overall responsiveness of the web application. In case of insufficient numbers of contributing browsers, function invocation falls back to the same functions deployed on a cloud-based FaaS infrastructure.

The remainder of this paper is organised as follows:

- §2 introduces the main technologies behind EDGAR and discusses the underlying assumptions and threat model;
- §3 describes EDGAR's design, explaining the systems requirements, its orchestration and explains how trusted function execution in browsers is achieved;
- §4 gives details on how we implemented EDGAR as well as how developers can implement EDGAR functions; and
- §5 shows that EDGAR scales linearly with increasing numbers of clients, copes with unresponsive clients and generates lower costs than traditional cloud deployments.

2 OFFLOADING FUNCTIONS TO WEB BROWSERS

Here, we describe the three emerging technologies that form the basis of EDGAR: Peer-to-peer communication between browsers (§ 2.1) with WebRTC, browser-based computation with JavaScript and WebAssembly (§ 2.2), and trusted execution as provided by Intel SGX (§ 2.3). We conclude this section with defining our assumptions and threat model (§ 2.4).

2.1 Browser-to-browser Communication

In the past, browsers were unable to communicate directly with each other. Instead, communication was achieved by using the server-side web application as a relay; adding unnecessary latency and imposing a potential bottleneck. Web Real-Time Communication (WebRTC) [7] changes this with a collection of protocols that enable direct browser-to-browser communication. WebRTC is already supported by all major browsers and provides mature JavaScript APIs. In addition to audio and video streams, it offers

https://doi.org/10.24355/dbbs.084-202111031112-0

data channels to transmit arbitrary data. However, WebRTC cannot work entirely without servers. For connection establishment, a well-known *signaling server* is needed. Edgar uses WebRTC to invoke distributed functions directly from other browsers.

2.2 Computation in Web Browsers

Computation within web pages is possible since the appearance of JavaScript. However, the language itself has many shortcomings that mainly result from its weak type system. Therefore, JavaScript is increasingly being used as a compilation target for stricter programming languages such as TypeScript [8]. The newest advancement in terms of browser-based computing is WebAssembly, a platform-independent binary instruction format [9]. Its code is not designed to be written by programmers directly; instead, it is used as a compilation target. Mature support for compilation from C, C++, C# and Rust already exists, while support for other programming languages including Java, Go, and Python is currently being developed. Edgar uses the JavaScript and WebAssembly runtime V8 for function execution.

From a service provider's point of view, computations performed in browsers are untrusted. Browsers can return wrong, incomplete or no results at all. This problem is usually circumvented by validating the results on the server-side, which often involves recomputations [10]. Additionally, it has been shown that input validation happening on both client- and server-side can introduce vulnerabilities [11]. Furthermore, this approach leads to undesired code duplication between client- and server-side code; in the worst case in different programming languages.

2.3 Intel Software Guard Extensions

Starting 2015, Intel's consumer CPUs include the Software Guard Extensions (SGX), which enable the instantiation of Trusted Execution Environments (TEEs). Such a TEE protects the integrity of code and data and is called *enclave* in the context of SGX. All computations inside such an enclave are isolated from potentially malicious software components, including privileged code. Enclaves allocate an isolated memory region within the process's address space. Pages of this memory region are stored in a reserved memory region, called the Enclave Page Cache (EPC). SGX protects the integrity and confidentiality of all EPC pages using checksums and transparent memory encryption. Additionally, enclaves can be authenticated by a remote challenger in a process called *remote attestation*. Thereby, an enclave identity including the enclave's code and, optionally, user-defined data is authenticated. The challenger sends the signed report to a trusted attestation service that can confirm the trustworthiness of the corresponding enclave, i.e. whether it contains the expected code and is running on a genuine SGX platform.

Adapting Applications for SGX. Existing applications cannot be executed in SGX enclaves without additional measures. Due to the underlying threat model, certain operations like system calls are not allowed, as they might compromise enclave isolation. The straightforward way of enabling legacy applications (e.g. a WebAssembly runtime) to run in SGX enclaves is manual partitioning (e.g. [12]). However, this imposes an extensive effort for larger applications and only limited approaches for automated partitioning exist [13, 14]. Several research projects [15–17] already explored possibilities of running legacy applications without changing their code using shielded execution and library operating systems (OSs). Edgar uses such a library OS, namely SGX-LKL [17] to enable execution of JavaScript and WebAssembly inside SGX enclaves.

Availability and Alternatives. With SGX being available on many recent Intel CPUs, our implementation of Edgar naturally relies on it. Since Edgar's design is not tied to SGX, alternative TEE implementations such ARM TrustZone or Keystone [18] could be used (see §4). These, however, offer different security guarantees.

2.4 Assumptions and Threat Model

In an Edgar deployment, two entity types interact frequently: many *users* and a single *service provider*. The users want to use the web application provided by the service provider. In turn, the service provider wants to make use of the users resources to provision that very service. We expect users will support the usage of client-side resources for different reasons, depending on the actual application. In the simplest case, users will regard the web application itself as incentive and use it as is. For scientific or non-profit organisations, altruistic users will support applications that would otherwise not be possible, such as a volunteer computing system like SETI@home[2]. Additionally, Edgar enables users to indirectly pay for services by contributing to their provisioning. This keeps providers from displaying ads, charging fees or selling their users data to cover their expenses.

As client machines are outside the service provider's control, we consider them as untrustworthy and propose that providers offload sensitive computations or critical data only to clients equipped with a TEE. Furthermore, users do not trust the machines of other users. However, we assume the user to the trust service provider to perform a proper remote attestation (see § 2.3) with all peers. This assumption is reasonable, as a proper attestation is in the interest of the service provider who wants to ensure the integrity of the application. Finally, the resources provided by the users are still unreliable. Frequent disconnects have to be expected, as users might close their browsers, shut down their machines or have an unstable network connection. Since Edgar relies on Intel SGX, we assume a correct implementation of SGX in hard- and software as well as a properly working attestation service (see § 2.3). We are aware of side-channel attacks affecting SGX (e.g. [19–23]). These have either been fixed by updates of microcode[3], the SGX SDK[4]. Furthermore, research on side-channel mitigation has been conducted [24–26]. We expect work to continue in this direction and therefore consider them outside the scope of our work. Finally, in contrast to most SGX-related threat models, we explicitly do consider denial-of-service attacks in the distributed setting of Edgar. If parts of the application are subject to such attacks, execution can be transferred to other participants or the server-side fallback environment.

[2] https://setiathome.berkeley.edu/

[3] https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/overview.html

[4] https://software.intel.com/sites/default/files/180204_SGX_SDK_Developer_Guidance_v1.0.pdf

https://doi.org/10.24355/dbbs.084-202111031112-0

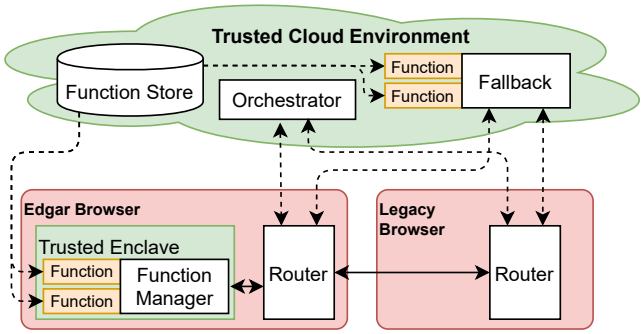


Figure 2: Architecture with one Edgar browser and one legacy browser. Trusted components are denoted in green, untrusted ones in red. Solid arrows show function invocations on other peers, while dashed arrows show more infrequent interactions such as fetching functions, communication with the orchestrator or invoking fallback functions.

3 DESIGN

In this section, we describe how we design Edgar. We start by detailing the system requirements in accordance with our threat model (§ 3.1), followed by a brief overview of Edgar in § 3.2. Furthermore, we explain how Edgar is deployed in a scalable manner in the cloud (§ 3.3) and on client-side (§ 3.4), give details on Edgar functions (§ 3.5), discuss which types of applications benefit most from Edgar (§ 3.6), and describe security-related design choices (§ 3.7).

3.1 System Requirements

A system for securely offloading functions to distributed, untrusted web browsers must fulfil the following requirements:

- R_1 *Scalability.* An Edgar application should scale linearly with increasing numbers of clients.
- R_2 *Isolation.* Function execution must be isolated from the executing host, i.e. the code delivered to clients needs to be unable to access or manipulate its environment.
- R_3 *Language agnosticism.* Edgar should support a wide range of programming languages for the developer to choose from; also to ease porting efforts for legacy applications.
- R_4 *Integrity and confidentiality.* Edgar should protect the integrity of all functions as well as the confidentiality of their in- and outputs, enabling to offload code that would traditionally be located on the server-side.
- R_5 *Availability.* All function calls issued to Edgar should eventually succeed, even if an insufficient number of peers participate or individual browsers misbehave.

In the following, we describe Edgar in a nutshell to explain how it fulfils these requirements.

3.2 Edgar in a Nutshell

Fig. 2 shows the architecture of Edgar with two clients; one Edgar browser (i.e. a browser with support for trusted function execution) and one legacy browser (i.e. a browser without TEE support). Edgar allows deploying web applications on distributed, usually untrusted browsers to reduce the centralised resources needed for operation. While this diminishes costs for service providers, it can also reduce load and response times for users. Clients can rely on nearby peers that have a certain functionality already loaded and initialised. All Edgar browsers that load an Edgar application automatically contribute to provisioning that very application, thus, increasing overall scalability (R_1). Furthermore, legacy browsers can still benefit from other Edgar browsers without contributing to it. However, activation of this feature is optional.

Following the paradigm of FaaS, Edgar applications consist of one or multiple *functions*. These allow a fine-grained, parallel distribution, which also increases scalability (R_1). Edgar functions can be developed using JavaScript and WebAssembly, providing two important properties: First, isolation from the host is guaranteed (R_2), because runtimes for both languages are designed as execution sandboxes. These are already used for executing untrusted code in browsers today. Second, many programming languages are supported (R_3), as both languages can act as compilation targets (see § 2.2). This eases porting existing FaaS applications to Edgar.

When an Edgar browser loads an application, the middleware first contacts the *orchestrator*. It has a global view over all participating browsers and can decide which set of functions the newly connected browser should *install* locally, based on an application-specific policy (e.g. highest priority for latency-sensitive functions). Additionally, the orchestrator is used by new clients for *peer discovery*; it continuously pings the peers and measures the response times of peers to enable discovery based on proximity. For installation, function code is first fetched from a cloud-hosted *function store* and then deployed locally in a TEE based on Intel SGX (see § 2.3). This *enclave* protects the integrity of function code and confidentiality of in- and outputs (R_4). Optionally, functions can be delivered in an encrypted format to gain code confidentiality. This way, application logic that usually requires a trusted server environment can be offloaded to clients. Installed functions can then be *invoked* by other browsers. The *router* component located on every client forwards the application's *function invocation request* to clients that have that function installed. Note, that this can also be the same client that issued the request. All functions are additionally deployed on a FaaS infrastructure we refer to as the *fallback*. This allows bootstrapping an Edgar application without enough peers being available (R_5). Furthermore, this generates negligible costs when enough peers are available.

Providers of an Edgar application should distribute the workload evenly across clients, so that users cannot differentiate between only using an application or providing resources to its provisioning. We argue that users implicitly accept providing resources, because their incentive is to use the application. A weaker form of this concept is also common practice today, as most users do not inspect the client-side code before visiting a website. Furthermore, Edgar allows limiting the resource usage (see § 3.4) and stops allocating resources as soon as the application of interest is closed. Especially for applications of non-profit organisations or scientists, altruistic users might voluntarily donate resources. For other applications, in conjunction with trusted resource accounting [27, 28], premium models can be used to create additional incentives, e.g. by offering ad-free services.

https://doi.org/10.24355/dbbs.084-202111031112-0

3.3 Edgar's Cloud Components

Here, we describe Edgar's cloud-based components and discuss their scalable deployment.

Orchestrator. To operate, Edgar needs to keep track of available peers. In theory, a fully distributed approach (e.g. with Edgar peers deploying a Distributed Hash Table (DHT) [29]) would be possible. However, to allow bootstrapping especially for smaller applications, this would require peers that also contribute resources when not using any Edgar application. Therefore, Edgar deliberately uses a centralised component operated by the service provider for orchestrating peers: the *orchestrator*. The orchestrator keeps track of all available peers for two reasons. First, it can instruct them to load a set of functions. Second, the stored information is used to enable peer discovery. Before function deployment, the service provider adds a *distribution factor* and *weight* to each function. The higher the distribution factor, the more will the corresponding function be installed. The weight represents the *expensiveness* of functions, reflecting how many resources it uses. When Edgar is initialised on a new client, it connects to the orchestrator to announce its availability and *machine size*, reflecting its computing power. This value can be determined by the browser's user agent, static hardware information, or a benchmark executed before connecting.

Subsequently, the orchestrator identifies the *least represented function* fitting the connecting client according to distribution factor and weight, and instructs it to install this function. This process is repeated, as long as the cumulative weights of all functions does not exceed the machine size. The orchestrator stores the mapping of which functions are distributed to which peers in order to share a subset of this mapping with new clients. When an Edgar application needs to invoke a function, the Edgar client first sends a discovery request to the orchestrator containing the unique function name. The orchestrator responds with a list of peers that have already loaded this function. This list is limited by proximity of peers, which is determined by latency estimation performed by the orchestrator. Additionally, using the IP addresses of clients for geotargeting can give a rough estimation of their location (e.g. from which region or country they connect). Edgar clients cache this information to relieve the orchestrator from requests.

Function Store. The function store is a HTTP server serving static assets. These consist of code in form of JavaScript or WebAssembly that makes up the application using Edgar. In order to assert trust, all code is integrity protected and can optionally be encrypted.

Fallback. For every Edgar application, it is essential that a sufficient number of peers is available. When bootstrapping Edgar applications, all functions are additionally deployed as a server-side fallback. As soon as enough peers are available (per region), the fallback does not need to handle any function invocations.

3.3.1 Proximity-based Deployment. In order to deploy Edgar's centralised components, well-known approaches can be used. Since code of Edgar functions is static and integrity protected, deploying the function store via an untrusted CDN can significantly improve scalability and response times. To prevent the orchestrator from becoming a bottleneck, is is possible to replicate multiple instances of it in different regions. Since Edgar clients discover the orchestrator based on proximity, this approach is transparent to clients. Finally, deploying the fallback functions on a traditional FaaS infrastructure achieves good scalability without generating high costs as it only receives invocations sporadically, especially when bootstrapping a new application.

3.4 Edgar on Client-side

The client-side components of Edgar are separated into a trusted and untrusted part. This is achieved by adding a second execution engine to the Edgar browser. This engine is protected by an SGX enclave and therefore trusted. While both engines are capable of executing JavaScript and WebAssembly code, the trusted one only executes signed (and optionally encrypted) code. Functions are executed in the trusted part, which can happen on behalf of a local or remote function invocation. In the untrusted part, the router component uses information obtained from the orchestrator to delegate function invocations.

Starting Edgar. As instructed by the orchestrator, the client downloads functions from the function store and calls into the trusted part to forward the code. In the trusted part, the *function manager* verifies the signature and decrypts the code if necessary; then it installs the functions.

Connection Management. After requesting peers in proximity from the orchestrator, the router preemptively establishes WebRTC connections to all of them to decrease latencies for first-time calls. To reduce call latencies, the first invocation is issued to whichever connection is established first. After this invocation, clients continuously measure the response times to choose low-latency peers for future calls. Since the peer's current load and connection quality also influences these measurements, Edgar automatically prioritises non-overloaded peers with a good connection. During application runtime, it is not uncommon for a peer to disappear, e.g. due to network issues or users closing their browsers. Therefore, peer connections are terminated upon close requests or if WebRTC keep-alive messages are not responded to. When many peers disconnect, the Edgar client asynchronously issues new requests to the orchestrator.

Limiting Resource Usage. The resources consumed by Edgar functions can be limited by kernel- (e.g. `cgroups`) or browser-based solutions (e.g. Opera GX[5]). While this can prevent overloading of clients, it has implications for service providers as users can now adjust their settings to only contribute minimally. In combination with trusted resource accounting [27, 28], service providers can incentivise (e.g. add-free services, exclusive features, or digital goods) users to provide resources above certain limits or even exclude them from the application.

3.5 Edgar Functions

An Edgar application can instruct the function manager to invoke a function by passing its name and parameters. The function manager directly invokes it if it is installed locally; otherwise, a remote peer or ultimately the fallback is contacted. For that, the router sends an *invocation request* via WebRTC, including the encrypted

[5] https://www.opera.com/gx

https://doi.org/10.24355/dbbs.084-202111031112-0

application category	example app	example function	data source	data sink	data size	suitability
collaboration tool	Etherpad	change set merging	peer	peer	small	●●●
web game	Agar.io	anti-cheat	peer	peer	small	●●●
digital assistant	Almond [30]	intent classification	peer	cloud	small	●●●
private web search	Cyclosa [31]	query forwarding	peer	cloud	small	●●●
upload preprocessing	Twitter	image scaling	peer	cloud	medium	●●○
recommendation system	YouTube	retrieve recommendations	cloud	peer	small	●●○
video conferencing	Jitsi	stream processing	peer	peer	large	●●○
p2p video streaming	PrivaTube [32]	stream processing	peer	peer	large	●●○
streaming service	Netflix	video transcoding	cloud	cloud	large	●○○

Table 1: Suitability of different web application categories for Edgar. Categories showcased in this paper are highlighted.

parameters. After the receiving peer's function manager decrypts the parameters, the function is invoked locally and the encrypted result is returned to the client. In case no peer with the appropriate function is available, the fallback is used. For the application, function invocation is transparent, regardless of where the function is ultimately invoked. The router manages a *peer rating* for every peer by periodically measuring the response time to all connected peers. Using the peer with the lowest response time allows Edgar to find the closest peer available and relieve potentially overloaded peers.

Function State. Edgar functions are –in the best case– stateless, e.g. they do not rely on persistent data shared with other functions. However, function code can access existing storage services for sharing data. Edgar also supports direct access to services that are usually part of the back-end (e.g. a database, blob storage or key-value store) as described in § 3.7. Additionally, Edgar supports *local function state* by allowing functions to persist data in browsers by using standard web APIs.

3.6 Edgar Applications

The application parts of an Edgar application that should be offloaded must consist of self-contained functions; hybrid applications where only parts run on top of Edgar are possible. Since this matches the programming model of FaaS, an existing FaaS application can also run on Edgar. However, as for FaaS, certain types of applications are more suited for an Edgar deployment than others. For example, Edgar is especially suited for applications that do not solely depend on centrally stored data. Additionally, the size of processed data is important, as well as whether it is processed locally on Edgar peers or centrally. Furthermore, the application's latency sensitivity and computation-intensiveness also influence its suitability for Edgar. Table 1 lists several example application categories, showing their main data sources (excluding initialisation) and data sinks. Here, *peer* means that data is generated or processed on Edgar peers, whereas *cloud* means that data is loaded from or stored at central locations. The table distinguishes between data sizes from small (<10 MB) over medium (<500 MB) to large (≥500 MB). We rate the suitability of applications for Edgar with excellent (●●●), good (●●○) or poor (●○○). For example, applications that process small amounts of data from peers are excellent use cases for Edgar, such as collaboration tools, games, digital assistants or private web search. We still assume good suitability, if applications process medium data sizes generated at peers, e.g. for preprocessing uploads such as images or videos for social networks. The same applies for applications where small amounts of cloud-based data is processed, such as in a recommendation system. When processing large amounts of data, we rate the application to still have good suitability, if the data is produced and consumed on Edgar peers, as in peer-to-peer video conferencing streaming. However, when a lot of cloud-based data is processed and then again stored in the cloud, we see poor suitability, e.g. video transcoding.

3.7 Security

Service providers of Edgar applications need to ensure that only SGX-enabled clients execute functions and that these clients execute the correct code. To achieve that, remote attestation is used (see § 2.3). While loading the web page, a remote attestation request is sent to the service provider, containing (i) a *quote*, which identifies the enclave's code as well as initial data and can be forwarded to the Intel Attestation Service (IAS); and (ii) an ephemeral public key which is generated during enclave startup and cryptographically bound to the quote. Upon retrieval, the service provider verifies that the enclave identity is known, i.e. that the client is running a genuine Edgar enclave containing the expected code. Then, the quote is forwarded to the IAS; a positive reply indicates that the enclave was started on a genuine SGX-capable platform. The remote attestation process finishes with the service provider using the enclave's public key for encrypting a symmetric key called the *script secret* with the enclave's public key. The script secret is unique for every web application and can be updated regularly. Edgar clients use this secret to verify Message Authentication Codes (MACs) of (i) JavaScript code embedded in the HTML of the web application; (ii) JavaScript or WebAssembly code retrieved from the function manager; or (iii) messages exchanged between enclaves. Additionally, the script secret can be used for decrypting any of these assets, if confidentiality protection is enabled. By using MACs instead of signatures, non-repudiation of messages is not fulfilled. However, since the script secret is only distributed to attested enclaves, this is no concern.

Attestation Between Peers. Edgar enclaves primarily communicate with other enclaves, e.g. for function invocation. All communication is integrity protected and encrypted using the script secret. The service provider is considered trusted (see § 2.4) and only exposes the script secret to trustworthy enclaves. Therefore,

https://doi.org/10.24355/dbbs.084-202111031112-0

EDGAR establishes trust between peers using the script secret because (i) only trustworthy enclaves are able to decrypt peer-to-peer messages, and (ii) an enclave receiving such a message can be sure that the communication partner has undergone a successful remote attestation with the service provider. Using this *implicit attestation*, no further actions after connecting to a peer need to be performed to establish a secure channel.

Generic TLS Proxy. Most web applications interact with back-end systems for server-side application logic or storage (e.g. blob storage, databases or key-value-stores). These systems are usually deployed behind web servers, should not interact with clients directly, and often have no support for encrypted access. Therefore, EDGAR includes the *generic TLS proxy*, which is deployed in front of back-end systems and uses a special Certificate Authority (CA) to enable *implicit remote attestation* between the proxy and connected enclaves. The CA issues client certificates after the remote attestation process described before. Accepting only certificates from this CA, the proxy employs mutual authentication to ensure only genuine EDGAR enclaves have access.

4 IMPLEMENTATION

EDGAR's function store is implemented as an HTTP server in TypeScript, the orchestrator is written in TypeScript as well. The signaling server uses *PeerJS*[6] for WebRTC signaling. We implement the orchestrator and signaling server in the same process; as this eases communication between these components, e.g. for notifications when clients disconnect. The fallback (see § 3.3) is deployed on *AWS Lambda*[7] and *Cloudflare Workers*[8]. EDGAR's generic proxy (see § 3.7) is based on the universal TLS tunnel *stunnel*[9] and is implemented as an HTTP server in Node.js. The client-side part of EDGAR is written in TypeScript as well, also uses PeerJS and connects to the orchestrator via WebSockets. Although we envision support for trusted execution being directly available as a browser API, we resort to implementing our prototype as a browser extension based on the *WebExtensions* API[10]. The extension communicates via native messaging[11] with the *backend*, which is the V8[12] runtime on top of SGX-LKL [17] enabling trusted execution of JavaScript and WebAssembly. The only SGX-specific component of EDGAR is this backend. Replacing it with an implementation based on a different TEE platform such as ARM TrustZone[13], AMD-SEV[14] or Keystone [18] is possible. However, since these technologies offer different security guarantees, some assumptions have to be reconsidered when porting EDGAR.

Code Examples. In the following, we showcase the convenience of EDGAR function implementation for developers. For the sake of simplicity, we use two functions: `add(a,b)` and `sub(a,b)` which add or subtract their inputs `a` and `b`. In this example, `add` is a function written in pure JavaScript, while `sub` is as WebAssembly function originally written in C. Listing 1 shows the C code of `sub` which is compiled into a WebAssembly module using Emscripten [33]. The `EMSCRIPTEN_KEEPALIVE` macro makes the function an *exported function* of the WebAssembly module (i.e. it is callable from JavaScript) and additionally prevents inlining or removal by the compiler. Listings 2 and 3 shows the JavaScript code needed for `add` and `sub`. In this case, the `add` function does not need any initialisation, but more complex functions would initialise libraries here (e.g. Tensorflow, see § 5.2). In contrast, the JavaScript wrapper for `sub` needs to fetch the corresponding WebAssembly module from the function store, decrypt and instantiate it via the corresponding JavaScript API[15]. After that, the wrapper can call the

```c
#include <emscripten/emscripten.h>

//exported function
int EMSCRIPTEN_KEEPALIVE sub(int a, int b) {
    return a - b;
}
```

Listing 1: C Code for WebAssembly `sub` function

```javascript
// no wasm module needed

function add(a, b) {
    return a + b;
}

async function __init() {
    //nothing to do in this case
}

return {
    init: __init,
    func: add
};
```

Listing 2: Pure JavaScript code for `add` function.

```javascript
let wasm = null;

function sub(a, b) {
    return wasm.exports['_sub'](a, b);
}

async function __init() {
    //fetch, decrypt and
    //instantiate WASM module
}

return {
    init: __init,
    func: sub
};
```

Listing 3: Wrapper code for WebAssembly `sub` function.

[6] https://peerjs.com/
[7] https://aws.amazon.com/lambda/
[8] https://workers.cloudflare.com/
[9] https://www.stunnel.org/
[10] https://developer.mozilla.org/docs/Mozilla/Add-ons/WebExtensions
[11] https://developer.mozilla.org/docs/Mozilla/Add-ons/WebExtensions/Native_messaging
[12] https://v8.dev/
[13] https://developer.arm.com/ip-products/security-ip/trustzone
[14] https://developer.amd.com/sev/
[15] https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate

https://doi.org/10.24355/dbbs.084-202111031112-0

exported function `_sub` (which is preceded by an underscore according to Emscripten convention). The Edgar middleware expects each function definition to return an object containing a mapping to the callable function (`func`) and the initialisation function (`init`). Using this object, the local function manager can initialise the function when loading it and subsequently invoke the corresponding function. For invoking the function, the client-side code simply calls the `add` or `sub` function locally; they resolve to stubs generated by Edgar that create and handle the invocation request. Deploying these Edgar functions as fallback on state-of-the-art FaaS offerings such as AWS Lambda or Cloudflare Workers is easily possible since both also use Node.js as JavaScript/WebAssembly runtime.

5 EVALUATION

In this section, we evaluate different aspects of Edgar and its underlying technologies. We start with evaluating Edgar's latency impact (§ 5.1). Then, we describe use case applications (§ 5.2). We continue with evaluating end-user latency and service costs (§ 5.3) as well as the scalability of Edgar (§ 5.4). Our last evaluation (§ 5.5) covers the impact of unresponsive peers on Edgar. Finally, we conclude this evaluation section by discussing attacks in § 5.6.

5.1 Latency Impact of Edgar

The end-to-end latency between to Edgar peers experiences delays from different sources such as (i) overhead of enclave transitions; (ii) overhead due to copying data to/from enclave memory; (iii) additional overhead induced by SGX-LKL; and (iv) overhead added by the Edgar browser extension. To evaluate the overall latency impact on Edgar, we perform a round trip measurement using two machines equipped with an SGX-capable Intel Core i7-6500U CPU connected via a 1 GBit switch. On both machines, we run the Chromium browser with our extension installed and exchange messages via WebRTC synchronously, i.e. the next message is sent when the reply for the first one is received. We send 100 messages for payload sizes from 16 bytes to 1 MB. To compare WebRTC via Edgar with plain WebRTC, requests and replies are either routed through the Edgar browser extension and the attached SGX enclave or not. Furthermore, we add artificial network delays of 10 ms and 50 ms using `tc netem` to simulate different distances between peers. Fig. 3 shows the average latencies for one message exchange. We ignore the reproducible outlier for WebRTC via Edgar with a 10 ms delay for a 256 KB payload, as we suspect a bug in either Chromium or Edgar's browser extension. For smaller payloads up to 4 KB, the latencies are constant, with plain WebRTC adding 1−2 ms and Edgar additionally adding 3−4 ms. For larger payloads up to 1 MB, the latencies increase; Edgar adds up to 250 ms due to larger buffers that are copied from the website via the browser extension into the SGX enclave and back. For no artificial delay (i.e. for sub millisecond latencies), the relative latency overhead of Edgar is high: 220% for small and 68% for large payloads. For medium artificial delays (10 ms), Edgar's average latency overhead for all payload sizes is 22%. Finally, for larger artificial delays (50 ms), the average latency overhead is 6% for small payloads and up to 20% for larger payloads. In summary, Edgar adds a small delay in most cases. While we observe a large relative overhead for local networks, it does only have a small effect when latencies

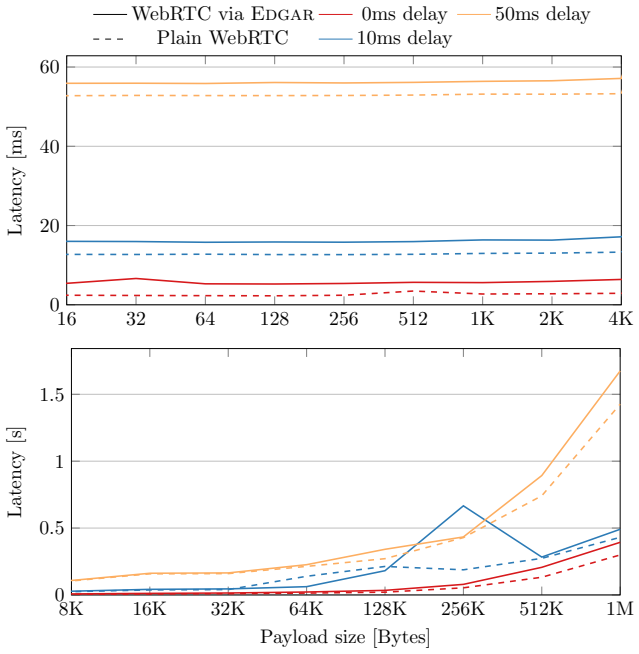


Figure 3: Average latency impact of Edgar for different network delays.

are more realistic. Note that the browser extension itself induces the highest overhead of 1.1 ms in our non-optimised prototype. If trusted execution would be natively supported by browsers, the only overhead would be the latency overhead of enclave transitions in the order of microseconds [34, 35].

5.2 Use Cases

To evaluate Edgar, we implement two realistic web-based use cases with different suitabilities from our discussion in § 3.6. First, a digital assistant based on machine learning which performs intent classification of text inputs and, second, a movie recommendation system. We intentionally choose these two quite different use cases. While the digital assistant processes data produced at the peers, the recommendation system depends on a centralised database. However, both process relatively small amounts of data.

Digital Assistant. Our digital assistant is a chat bot designed to be embedded into websites to interact with its users. It applies natural-language understanding (NLU) on text messages with a pre-trained model of approximately 5 MB to identify intents. We regard this application a good example for Edgar, because (i) a pre-trained model include secret data that motivates the application of trusted execution; and (ii) offloading to nearby browsers is beneficial, if the model is already loaded by the.

Therefore, this is a good example where offloading to nearby browsers is beneficial, because the peer has already loaded the model. The assistant supports the following seven intents: `greet`, `bye`, `affirmative`, `negative`, `wtf`, `playMusic`, and finally `addEventToCalendar`. The first four intents are used for controlling the conversational flow, while insults or out-of-context messages are classified as `wtf`. The two last intents trigger actual

https://doi.org/10.24355/dbbs.084-202111031112-0

function deployment	response time (ms)	billed durations (ms)	cost for 10M invocations
A: EDGAR same netw.	196 - 285	none	$0
B: EDGAR same city	246 - 329	none	$0
C: EDGAR same region	249 - 330	none	$0
D: Lambda 3008 MB	370	300 - 400	$150 - $200
E: Lambda 2048 MB	378	300 - 400	$100 - $133
F: Lambda 1024 MB	694	600 - 700	$150 - $175

Table 2: Response times, most billed durations in AWS Lambda and invocation costs for digital assistant use case.

actions; either streaming songs or creating calendar entries. We assume the model itself and its inputs are confidential, classification can therefore not be offloaded to clients without EDGAR. Our bot is implemented in TypeScript, based on the Aida chat bot[16]. It uses Tensorflow.js[17], which also supports training and classification in WebAssembly. The intent classification function consists of 83 lines of code and can be deployed on EDGAR peers as well as on the AWS Lambda FaaS platform. When invoked, the function fetches model data from S3, loads it into Tensorflow, then performs intent classification and finally returns the detected intent together with a confidence value between 0 and 1. The model is only fetched once and cached for future invocations.

Movie Recommendation System. The movie recommendation system allows its users to browse or rate movies. Additionally, they can list recommended movies, while recommendations are based on their and other users' ratings. We implement it using TypeScript and *Vue.js*[18]. It contains nine individually distributable functions, which allow users to log in to the application, browse movie titles by genre, view and modify movie ratings, and retrieve recommendations or information about specific titles. All functions obtain information from a central MySQL database, which is securely accessible through EDGAR's TLS Proxy (see § 3.7).

5.3 Response Times and Invocation Costs

Here, we want to show that deploying functions with EDGAR can actually reduce the response times experienced by end-users as well as the cloud costs for service providers. Since EDGAR enables offloading code that is usually server-side, we compare deployments of the classification function of our digital assistant use case on EDGAR and AWS Lambda. To facilitate different machine types acting as EDGAR peers, we install the classification function both on a laptop with an i7-6500U CPU and a desktop machine with an i7-6700 CPU.

We deploy one invoking and one executing EDGAR peer in the following configurations: (i) two EDGAR peers in the same university network (config A); (ii) two EDGAR peers in the same city, the invoking peer in a university network, the executing peer in a home network (config B); (iii) two EDGAR peers in the same region, the invoking peer in a university network, the executing peer in a home network in a different city in the same region (config C). The home networks are connected to the internet via DSL. In all configurations, the orchestrator is located in the university network, but not part of the latency measurements. We use Node.js processes instead of browsers to perform the measurements. This only minimally influences the results, as the same execution engine (V8) is used and the latency added by the EDGAR browser extension and SGX enclaves is negligible (see § 5.1).

We deploy the classification function in the AWS region closest to our university, which is *eu-central-1* (Frankfurt). In Lambda, developers define the allocated memory per function deployment from 128 MB to 3008 MB in 64 MB steps, which also linearly increases vCPU credits available to the function. We deploy the function in the following configurations: 3008 MB allocated memory, which is the maximum (config D); 2048 MB allocated memory (config E); and 1024 MB allocated memory (config F). Since our function consumes approximately 500 MB of memory, execution times become unacceptable for lower memory amounts: 1 second at 512 MB, up to 10 seconds at 128 MB at higher costs.

In all configurations, we use a machine in our university network for invoking the function and reporting the response time. We invoke the digital assistant's classification function 100 times with randomly chosen sentences from a test data set[19] consisting of 1500 sentences with lengths between 3 and 141 characters. We report the average response time experienced by end-users in Table 2. For the EDGAR deployments, we measure response times between 196 and 249 ms for desktop peers and between 285 and 330 ms for laptop peers depending on placement of peers. For AWS Lambda, we observe response times between 370 ms and 694 ms depending on the configured memory for the FaaS environment. This results in response times for EDGAR ranging from 11% (C laptop/D), over 13% (B laptop/E) and 35% (B desktop/E) up to 410% (A desktop/F) shorter than for a deployment on AWS Lambda.

Furthermore, we estimate costs for invocations in the AWS Lambda deployments. Function invocations are billed $2 per 10M request plus a fixed amount per 100 ms duration, rounded up to the nearest 100 ms. This fixed amount varies, depending on the AWS region and the allocated memory[20]. Table 2 shows the most billed durations by Lambda per configuration and the resulting estimated costs for 10M invocations for different deployments. While function invocation on EDGAR peers does not generate direct costs for service providers, an AWS Lambda deployment generates cost between $100 and $200 per 10M requests. To set this into context, we envision a web application of 10 (possibly chained) functions with one million daily active users, each invoking the functions 10 times per day. On AWS Lambda, this application would generate yearly costs of up to $730,000. In comparison, placing the EDGAR orchestrator on a reasonably large VM[21] in all 20 AWS regions would only generate costs of $24,000 per year. As functions only allocate relatively small amounts of storage, costs for a function store deployed on AWS S3 would be negligible[22].

[16] https://github.com/rodrigopivi/aida/
[17] https://www.tensorflow.org/js
[18] https://vuejs.org/
[19] https://aida.dor.ai/models/dataset_testing.json
[20] https://aws.amazon.com/lambda/pricing
[21] t3a.xlarge (4 vCPUs, 16GB RAM, 5Gbit/s) ~$100/month
[22] approx. $0.025 per GB per month, $0.0055 for 1000 write requests, $0.0004 for 1000 read requests, see https://aws.amazon.com/s3/pricing/

https://doi.org/10.24355/dbbs.084-202111031112-0

In summary, these observations show that users of Edgar can experience significantly lower response times compared to a traditional FaaS deployment, while also generating lower costs for service providers.

5.4 Scalability

An additional advantage of Edgar is that scalability of web applications can be achieved by leveraging client-side resources that implicitly scale with increasing numbers of users. To evaluate this, we take two different functions from our movie recommendation system (see § 5.2). The first function `getAllGenres` issues queries to the database and returns a list of all genres in the database. It is used as an example for an I/O intensive function not performing significant computation. The second function `getRecommendations` executes a nearest neighbour algorithm for the user's ratings and 100 other users' ratings to find recommendable movies. Although it also fetches user ratings from the database, most of its computation is used for the algorithm; it is, therefore, an example for a computation intensive function. We provision different AWS instance types (*t3.micro* for peers and *t3.medium* for servers) and deploy one server instance for Edgar's signaling server, orchestrator, function store, database and web server in the *us-east-2* region. Additionally, multiple Edgar peers are provisioned across 10 AWS regions[23]; again we use Node.js processes instead of browsers, because AWS does not support SGX yet. We install the two Edgar functions on all peers and provision a special peer called *measurer* with no functions installed in the *us-west-1* region. Note that no clients share a region with the database server for simulating realistic latencies. However, we intentionally let one regular peer and the measurer share the same region to represent nearby peers. In this measurement, we compare three configurations: (i) 1 peer close to the measurer; (ii) 20 peers, i.e., 2 peers per region; and (iii) the baseline, where functions are invoked traditionally on server-side via HTTP. We let the measurer issue invocation requests to Edgar peers at fixed rates and measure the response time, excluding connection establishments.

First, we let the measurer invoke the I/O intensive function `getAllGenres` at fixed throughputs from 1 to 1400 requests per second; in steps of 100. We stop measuring if the response time exceeds 1000 ms because this is a clear sign of the system being saturated. Fig. 4 shows the average call response times in correlation to the throughput for this function. We see that invoking functions through Edgar induces an up to 32% higher response time due to an additional network hop between the measurer and the database server. The configuration with one peer achieves stable response times for up to 538 requests per second, which is a 39% lower throughput than the baseline. Here, the single peer cannot handle as many requests as the more powerful server machine and becomes a bottleneck. When looking at the configuration with 20 peers, we see a higher throughput of up to 1,264 requests per second, 1.67× higher than the baseline. This setup scales until the capacity of the centralised database server is saturated.

Second, the measurer invokes the computationally intensive function `getRecommendations`. We use the same requests rates as before, but add measurements between 1 and 25 requests per second. Fig. 4 shows the average response time depending on the throughput. Compared to the previous measurement, we report (i) an overall lower throughput because the benchmark is CPU bound, and (ii) an overall higher response time, because the functions take longer to complete. Again, we see a higher response time due to one additional network hop. The configuration with 1 Edgar peer performs similarly to the baseline, as both achieve up to 24 requests per second. For more peers, Edgar achieves up to 468 requests per second, which is 19.5× the baseline and the single peer configuration; showing that Edgar scales linearly with increased numbers of peers.

To summarise, these two measurements show that the performance of Edgar scales with increasing numbers of peers because more resources become available for offloading. We also see that computation intensive functions are better candidates for offloading. Finally, in contrast to the evaluation of the digital assistant use case (see § 5.3), Edgar peers observe higher response times, because this recommendation system use case contains a centralised database, resulting in an additional network hop.

[23] specifically, we use us-east-1, us-west-1, us-west-2, ca-central-1 , sa-east-1, eu-central-1, eu-west-1, eu-west-2, eu-west-3 and eu-north-1

dwell time d	RTT r	$P_{unresp.}$ for workload durations (w)				
		1 ms	10 ms	100 ms	1 s	10 s
1 min	10 ms	0.01%	0.03%	0.18%	1.68%	16.68%
	100 ms	0.09%	0.10%	0.25%	1.75%	16.75%
	400 ms	0.34%	0.35%	0.50%	2.00%	17.00%
20 min	10 ms	0.00%	0.00%	0.01%	0.08%	0.83%
	100 ms	0.00%	0.01%	0.01%	0.09%	0.84%
	400 ms	0.02%	0.02%	0.03%	0.10%	0.85%
40 min	10 ms	0.00%	0.00%	0.00%	0.04%	0.42%
	100 ms	0.00%	0.00%	0.01%	0.04%	0.42%
	400 ms	0.01%	0.01%	0.01%	0.05%	0.43%

Table 3: Probabilities of unresponsive peers due to different dwell times, round-trip times and workload durations.

5.5 Unresponsive Peers and Reconnections

In a real-world Edgar application, peers are expected to become unresponsive due to multiple reasons: (i) users leaving the websites (e.g. closing their browser or specific browser tabs); (ii) external reasons such as network issues; or (iii) malicious peers that do not respond to invocation requests. For the first class of events, we introduce a probability model to estimate the probability of peers being unresponsive $P_{unresp.}$, which is described in the following. We exclude the second and third class from our model, as such events are unpredictable. In these cases, peers would run in the timeout configured by the function developer and retry with a different peer. We consider two peers A and B, where peer A sends invocation requests to peer B. We introduce the following variables: d, the dwell time of peer B (i.e., the time the user spends on the website), the WebRTC round-trip time r between A and B, and w, the duration the workload needs to finish processing. Peer B will be unresponsive, if the invocation request is sent in the last $l = w + \frac{r}{2}$ seconds of the dwell time d. Assuming a uniform distribution of

https://doi.org/10.24355/dbbs.084-202111031112-0

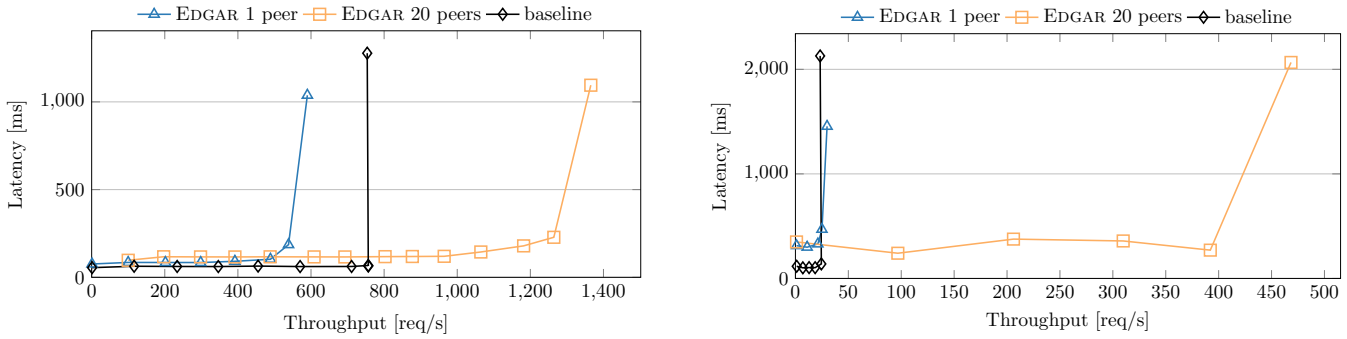


Figure 4: Throughput and latency of invocations of I/O intensive function (left) and CPU intensive function (right).

function invocations, we can estimate the probability $P_{unresp.}$ of an invocation request remaining unanswered with

$$P_{unresp.} = \frac{w + \frac{r}{2}}{d}$$

To calculate the probabilities, we assume the following realistic ranges for the introduced variables: Users spend between 70 seconds and 40 minutes on a website, depending on its type [36, 37]. Therefore, we assume 1, 20 and 40 minutes for the dwell time d. For WebRTC round trip times, we use the values of 10 ms, 100 ms and 400 ms reported in [38] for local, university and mobile networks. Since workload runtimes are application-specific, we assume values from 10 ms to 10 s for w. Table 3 shows the calculated probabilities for these ranges. We see, that for the most combinations, probabilities are well below 0.1% and only exceed 10% when the workload processing time ranges near the peer's dwell time (shaded cells). EDGAR's frequent WebRTC keep-alive messages further decrease these probabilities, as broken connections are detected earlier. For EDGAR, this means that websites with expected shorter dwell times are more suited for shorter workload executions, while websites with expected longer dwell times (such as social networks) are more capable of handling longer workload execution times.

We use the reported probabilities for the following experiment: We deploy a WebAssembly function performing an integer addition as a *Cloudflare Worker* (see §4) and on EDGAR. Choosing configured timeouts from 100 to 1,000 ms, we examine probabilities of 0.1%, 1% and 5%. These are induced by randomly dropping invocation requests with the given probability. A fully responsive peer (EDGAR baseline) and the Cloudflare deployment (FaaS fallback) are both unaffected by the configured timeout and act as baselines. We invoke every function 1000 times and report the average response time for a successful reply. The results in Fig. 5 show, that the EDGAR baseline performs best with an average response time of 4.5 ms, closely followed by 0.1% unresponsive peers with 4.9 ms average response time. All invocations to partly unresponsive peers stay below the average response time of the FaaS fallback of 62.7 ms. Our measured values only come close to this baseline if 5% of the peers are unresponsive and a relatively large timeout is configured. Combining these insights with the ones from our probability model, we conclude that an EDGAR deployment will be largely unaffected by unresponsive peers.

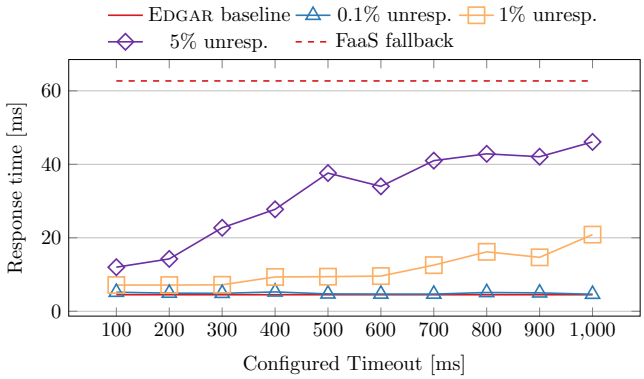


Figure 5: Response times for different shares of unresponsive peers and various timeouts.

5.6 Security Analysis

In the following, we discuss possible attacks against EDGAR according to our assumed threat model (see § 2.4).

Denial of Service (DoS) Attacks. Local attackers can stop or deny starting enclaves at any time. In typical SGX threat models DoS attacks are excluded, as there are no countermeasures. In EDGAR, such an attack only affects the attacker's machine and thus does not compromise the whole system. Whenever a client sends a request to a peer and does not receive an answer before a timeout occurs, it will retry with a different peer and ultimately use the fallback.

Unexpected Function Installation. The only entry point to the enclave, besides processing a message from a peer, is responsible for loading a function. An EDGAR function is shipped in a signed (and optionally encrypted) file containing WebAssembly or JavaScript code. Therefore, an attacker can only install functions that are signed by the service provider. Consequently, the attacker is only able to install functions defined by the service provider. However, as these functions' availability is not communicated to the orchestrator, other clients will never connect to this peer.

Peer Privacy Attacks. In EDGAR, IP addresses are exposed to other peers and the orchestrator. Both can use the information to log peer availability or draw conclusions regarding their location. An IETF draft [39] proposes to conceal IP addresses with dynamically

https://doi.org/10.24355/dbbs.084-202111031112-0

generated Multicast DNS names, thus protecting privacy. Edgar can use this as it is already implemented in Safari and has been announced for Chrome.

Overloading Peers. An attacker can flood a peer with invocation requests aiming to overload that peer's processing power. If an Edgar client observes an abnormally high number of requests from a single peer, the client terminates the connection to that peer and locally blacklists it.

Modified or Forged Messages. Due to the untrusted side of the system being responsible for accessing the network, an attacker is able to drop, modify or forge any network packets. All modified messages will be detected due to a wrong MAC and dropped. Message forging is not possible, as they are encrypted using the script secret (see § 3.7).

Replayed, Dropped or Out-of-Order Messages. In Edgar, all responses are verified to contain the same nonce used in the request to prevent replay attacks. Incorporating a monotonic counter in requests, enables the detection of messages being dropped or sent out-of-order.

6 RELATED WORK

In this section, we discuss research work related to Edgar.

Offloading to End-Users. Offloading computation to personal machines of end-users has also been explored in the context of *volunteer computing* [40–42]. However, research has also shown that contributors lose interest quickly [43]. Since Edgar focuses on interactive workloads that relate to the used web application, this is less of an issue for Edgar.

Offloading to Browsers. Several systems explore offloading computations or data to browsers, but do not make use of trusted execution at client-side: Akamai NetSession [44] is a commercially available CDN that is capable of offloading traffic to participating peers without support for offloading computations. Maygh [45] is a CDN consisting of browsers, offloading the delivery of static content to many clients. Similarly to Edgar, the system is based on WebRTC and includes one or more coordinators that manage the available peers. The main difference to Edgar is that Maygh can only deliver static content, but does not support the sharing of computation results, due to untrusted clients. Legion [46] and Pando [47] also apply WebRTC to distribute computations across web browsers, but do not consider untrusted clients.

Trusted Offloading to Browsers. Other works also combine trusted execution with web browsers. Our own previous work TrustJS [48] is the predecessor of the Edgar browser extension and also enabled trusted execution of JavaScript in web browsers. However, the early prototype of TrustJS was not suited for actual computations, since it is based on a slow interpreter without support for JIT or WebAssembly. Furthermore, it does not enable distributed applications as it has no support for orchestration or direct function invocation. Also, it is not usable anymore because it use a deprecated browser API. Fidelius [49] protects user inputs into web applications from a malicious operating system. It uses single-board computers with large Trusted Computing Bases (TCBs) to create a trusted path from input to output devices. Similarly to TrustJS, code is split into trusted and untrusted parts, each being executed in an SGX-protected JavaScript runtime. However, this runtime can only interpret JavaScript code at low performance and has no support for WebAssembly. Furthermore, Fidelius does not target distributed applications.

The goal of our previous work Cyclosa [31] is to protect personal data of web search users. It uses a peer-to-peer network in combination with SGX to obfuscate the user's query before sending it to the web search service. However, Cyclosa only supports one single application, which is web search. In fact, Edgar can be viewed as a sequel of Cyclosa allowing arbitrary applications: while Cyclosa cannot be used to implement other types of web applications, it could be implemented using Edgar (see § 3.6).

PrivaTube [32] is a browser-based CDN focused on video streams and uses SGX to establish trust in browsers. However, it considers neither direct communication between browsers nor computations.

7 CONCLUSION

In this paper, we presented Edgar, a novel middleware that allows service providers to distribute a web application over its current users' machines. Thus, service providers can save costs while users of such applications can contribute to its provisioning, e.g. in exchange for ad-free services. Edgar establishes trust into browsers by applying trusted execution technology to offload function execution securely. To improve response times, functions are directly invoked on nearby browsers using peer-to-peer communication. In our evaluation, we showed that Edgar copes well with unresponsive peers, generates lower costs and scales linearly with increasing numbers of participants.

ACKNOWLEDGMENTS

This work has received funding from Intel Corporation in the scope of the *TFaaS* research project.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
[2] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally Distributed Content Delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.
[3] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai Network: A Platform for High-Performance Internet Applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.
[4] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson, "Cost-Benefit Analysis of Cloud Computing Versus Desktop Grids," in *IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12, IEEE, 2009.
[5] J. M. Hellerstein *et al.*, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.
[6] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of client-side web (in)security," in *26th USENIX Security Symposium (USENIX Security)*, pp. 971–987, USENIX Association, 2017.
[7] W3C WebRTC Working Group, "WebRTC 1.0: Real-time Communication Between Browsers." https://www.w3.org/TR/webrtc/, 2021.
[8] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *European Conference on Object-Oriented Programming (ECOOP)*, pp. 257–281, Springer, 2014.
[9] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web Up to Speed With WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 185–200, ACM, 2017.
[10] K. Vikram, A. Prateek, and B. Livshits, "Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution," in *Proceedings of the 16th ACM*

https://doi.org/10.24355/dbbs.084-202111031112-0

Conference on Computer and Communications Security (CCS), pp. 173–186, ACM, 2009.

[11] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pp. 607–618, ACM, 2010.

[12] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper Using Intel SGX," in *Proceedings of the 17th International Middleware Conference*, pp. 1–13, 2016.

[13] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, *et al.*, "Glamdring: Automatic Application Partitioning for Intel SGX," in *USENIX Annual Technical Conference (USENIX ATC)*, pp. 285–298, USENIX Association, 2017.

[14] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves," in *29th USENIX Security Symposium (USENIX Security)*, pp. 505–522, USENIX Association, 2020.

[15] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, *et al.*, "SCONE: Secure Linux Containers With Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 689–703, USENIX Association, 2016.

[16] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *USENIX Annual Technical Conference (USENIX ATC)*, pp. 645–658, USENIX Association, 2017.

[17] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "SGX-LKL: Securing the Host OS Interface for Trusted Execution," *arXiv preprint arXiv:1908.11143*, 2019.

[18] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, pp. 1–16, ACM, 2020.

[19] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *European Symposium on Research in Computer Security (ESORICS)*, pp. 440–457, Springer, 2016.

[20] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-Grained Control Flow Inside SGX Enclaves With Branch Shadowing," in *26th USENIX Security Symposium (USENIX Security)*, pp. 557–574, USENIX Association, 2017.

[21] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom With Transient Out-Of-Order Execution," in *27th USENIX Security Symposium (USENIX Security)*, pp. 991–1008, USENIX Association, 2018.

[22] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T.-H. Lai, "SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution," *IEEE Secur. Priv.*, vol. 18, no. 3, pp. 28–37, 2020.

[23] K. Murdock, D. F. Oswald, F. D. Garcia, J. V. Bulck, F. Piessens, and D. Gruss, "Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble," *IEEE Secur. Priv.*, vol. 18, no. 5, pp. 28–37, 2020.

[24] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *24th Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2017.

[25] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves From Practical Side-Channel Attacks," in *USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 227–240, USENIX Association, 2018.

[26] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pp. 788–800, ACM, 2019.

[27] S. Tople, S. Park, M. S. Kang, and P. Saxena, "VeriCount: Verifiable Resource Accounting Using Hardware and Software Isolation," in *International Conference on Applied Cryptography and Network Security (ACNS)*, pp. 657–677, Springer, 2018.

[28] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting," in *20th International Middleware Conference (Middleware)*, pp. 123–135, IEEE, 2019. Rank A, 24% acceptance rate.

[29] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: A Public DHT Service and Its Uses," in *Proceedings of the SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 73–84, ACM, 2005.

[30] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam, "Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant," in *Proceedings of the 26th International Conference on World Wide Web (WWW)*, pp. 341–350, ACM, 2017.

[31] R. Pires, D. Goltzsche, S. B. Mokhtar, S. Bouchenak, A. Boutet, P. Felber, R. Kapitza, M. Pasin, and V. Schiavoni, "CYCLOSA: Decentralizing Private Web Search Through SGX-based Browser Extensions," in *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 467–477, IEEE, 2018.

[32] S. Da Silva, S. Ben Mokhtar, S. Contiu, D. Négru, L. Réveillère, and E. Rivière, "PrivaTube: Privacy-Preserving Edge-Assisted Video Streaming," in *Proceedings of the 20th International Middleware Conference*, pp. 189–201, ACM, 2019.

[33] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*, pp. 301–312, ACM, 2011.

[34] O. Weisse, V. Bertacco, and T. M. Austin, "Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 81–93, ACM, 2017.

[35] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves," in *Proceedings of the 19th International Middleware Conference*, pp. 201–213, ACM, 2018.

[36] C. Liu, R. W. White, and S. Dumais, "Understanding Web Browsing Behaviors Through Weibull Analysis of Dwell Time," in *Proceedings of the 33rd International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 379–386, ACM, 2010.

[37] F. Schneider, A. Feldmann, B. Krishnamurthy, and W. Willinger, "Understanding Online Social Network Usage From a Network Perspective," in *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pp. 35–48, ACM, 2009.

[38] S. Taheri, L. A. Beni, A. V. Veidenbaum, A. Nicolau, R. Cammarota, J. Qiu, Q. Lu, and M. R. Haghighat, "WebRTCbench: A Benchmark for Performance Assessment of webRTC Implementations," in *13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pp. 1–7, IEEE, 2015.

[39] J. U. Y. Fablet, J. de Borst and Q. Wang, "Using Multicast DNS to protect privacy when exposing ICE candidates." https://datatracker.ietf.org/doc/html/draft-ietf-mmusic-mdns-ice-candidates-02, 2021.

[40] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky, "SETI@home – Massively Distributed Computing for SETI," *Computing in Science & Engineering*, vol. 3, no. 1, pp. 78–83, 2001.

[41] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 4–10, IEEE, IEEE Computer Society, 2004.

[42] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons From Eight Years of Volunteer Distributed Computing," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–8, IEEE, 2009.

[43] O. Nov, D. Anderson, and O. Arazy, "Volunteer Computing: A Model of the Factors Determining Contribution to Community-Based Scientific Research," in *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pp. 741–750, ACM, 2010.

[44] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec, "Peer-Assisted Content Distribution in Akamai Netsession," in *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pp. 31–42, ACM, 2013.

[45] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram, "Maygh: building a CDN from client web browsers," in *Proceedings of the Eighth ACM European Conference on Computer Systems (EuroSys)*, pp. 281–294, ACM, 2013.

[46] A. van der Linde, P. Fouto, J. Leitão, N. Preguiça, S. Castiñeira, and A. Bieniusa, "Legion: Enriching Internet Services With Peer-To-Peer Interactions," in *Proceedings of the 26th International Conference on World Wide Web (WWW)*, pp. 283–292, ACM, 2017.

[47] E. Lavoie, L. Hendren, F. Desprez, and M. Correia, "Pando: Personal Volunteer Computing in Browsers," in *Proceedings of the 20th International Middleware Conference*, pp. 96–109, ACM, 2019.

[48] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "TrustJS: Trusted Client-side Execution of Javascript," in *10th European Workshop on Systems Security (EuroSec)*, pp. 1–6, ACM, 2017.

[49] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, *et al.*, "Fidelius: Protecting User Secrets From Compromised Browsers," in *Symposium on Security and Privacy (SP)*, pp. 264–280, IEEE, 2019.

https://doi.org/10.24355/dbbs.084-202111031112-0